expressed as \$08+\$FB. The result of this sum is \$103, which is \$03 as a single-byte number.

This kind of representation is known as *two's complement*: the complement of a single-byte number is formed by subtracting it from \$100. There is another representation known as *one's complement*, and the two are related in an interesting way. Consider this:

```
$05 = 00000101      binary
$FA = 11111010      one's complement
            +1
-------------------
$FB = 11111011      two's complement
-------------------
$05+$FA=$FF
$05+$FB=$00
```

The one's complement of a single-byte number is formed simply by complementing or negating each binary bit of the number. If one is added to this result, then the two's complement of the number is produced. A number and its one's complement always total \$FF, while a number and its two's complement always total \$00 (actually \$100). It is conventional then, in signed integer arithmetic, to regard the numbers from \$00 to \$7F as the positive numbers, (0 to 127) and \$80 to \$FF as the negative numbers (−128 to −1). If you compare the binary representations of these numbers you will notice that all the negative integers have bit 7 set, while in the positive numbers bit 7 is always reset. Accordingly, bit 7 is known as the *sign bit* of a signed number, and the carry flag of the processor status register is set or reset as a copy of bit 7 of the result of the last arithmetic or logical operation.

There is no easy way round this potentially confusing subject, and it simply has to be approached when you start doing signed arithmetic. Fortunately, once its implications are understood, it can be handled mechanically by rule-of-thumb methods. These methods, and the multiplication and division alogrithms, are the subject of the next instalment of the course.

## Answers To Exercises On Page 259

**1)** The following program reverses the order of the character string stored at LABL1:

**6502**

```
;
ORIGIN   ORG   $7000
LAST1    EQU   $0D
LABL1    DB    'THIS IS A MESSAGE'
TERMN8   DB    LAST1
;
BEGIN    LDX   #$FF
         LDA   #LAST1
         PHA
LOOP0    INX
         LDA   LABL1,X
         PHA
         CMP   #LAST1
ENDLP0   BNE   LOOP0
CLRSTK   PLA
;
BEGIN1   LDX   #$FF
LOOP1    INX
         PLA
         STA   LABL1,X
         CMP   #LAST1
ENDLP1   BNE   LOOP1
         RTS
```

In the 6502 version, the code between LOOP0 and ENDLOOP0 uses X-indexed addressing in a loop to load the characters one-by-one from LABL1, and push them onto the stack — having first pushed the ASCII value of the terminator character to mark the bottom of the stack. The last character pushed onto the stack is also the terminator, this time determined from its position as the last character in the string. This concludes the loop, and the terminate character on top of the stack is then cleared at CLRSTK.

The Z80 version uses IX in indirect addressing mode to load the accumulator from LABL1 onwards, and pushes not only the accumulator but also the flag register onto the stack. This means that the characters of the string at LABL1 are interspersed on the stack with successive values of the processor status register.

**Z80**

```
         ORG   $C000
LAST1    EQU   $0D
LABL1    DB    'THIS IS A MESSAGE'
TERMN8   DB    LAST1
;
BEGIN    LD    IX,LABL1-1
         LD    A,LAST1
         PUSH  AF
LOOP0    INC   IX
         LD    A,(IX+0)
         PUSH  AF
         CP    LAST1
ENDLP0   JR    NZ,LOOP0
CLRSTK   POP   AF
;
BEGIN1   LD    IX,LABL1-1
LOOP1    INC   IX
         POP   AF
         LD    (IX+0),A
         CP    LAST1
ENDLP1   JR    NZ,LOOP1
         RET
```

The code between BEGIN1 and ENDLP1 in both versions is a reflection of the previous loop and uses the same techniques, but this time pulling the character string off the stack in reverse order, and storing it at LABL1 onwards. The loop finishes when the terminator character is found at the bottom of the stack.

Notice how important it is to balance stack pushes and pulls, and that the most difficult part of the problem is deciding how to handle the extreme conditions — what to do at the start of the loops, how to terminate them, and what 'tidying-up' (if any) is then required.